Design in Practice

Rich Hickey

Objective - Demystify Design

not (just) going to the hammock

practice - *'what you do'*

concrete techniques with tangible outputs

demonstrable **progress** *'walk forward'*

activities you can make PM stories out of

thus make time for, throughout the dev process

not pleading for 2 weeks of nebulous 'hammock time' up front

valuable artifacts that make the effort evident

tips and techniques, not a formal method or anything highfalutin

Design

design - Latin for *'waiting to code'*

coding happens throughout

performing experiments

answering interim questions

why you want a language that supports exploratory programming
without being in a project building context

Design (cont.)

'mark out, a plan'

the emphasis in this talk is about supporting your (team's) reasoning **process**, not just the end-product blueprint-like design

writing down your thoughts helps you form them

techniques can guide your thinking and decision-making

reified/refined/shared concepts

onboarding/resumption

validation

eventually, documentation

Words

Choose good words, all the time
not about bikeshedding or premature marketing

precision in naming == precision in thinking *'before + cut'*

eschew nicknames, superheroes etc
not semantic/meaningful
give cover to fuzziness
don't track evolving thinking

be succinct *'gird/gather up'*
brief, clear and complete
not just concise *'cut off'*, or merely hinted at

More Words

use the dictionary (not just good for writing keynotes)
go right to the origins
- most useful/abstract semantics
- discover the composition within words

a good word later becoming 'wrong' could mean:

you've changed your mind w/o acknowledging it

you are drifting from your intentions

your thinking will evolve and your words (story titles etc) should also

Technique: Glossary

terms are inevitable in tech
valuable shorthand

don't presume a shared understanding

define, in one place

use uniformly and consistently

helps non-tech folks trying to follow along

when terms break, fix or abandon

Example: Glossary

	A	B
1	**Term**	**Meaning**
2	locality	A property of data: It is a measure of the distribution of datoms you need to find, across segments as seen from the perspective of one of the indexes. A measure of locality is the number of segments that need to be examined.
3	affinity	A strategy for assigning partitions, where you say that things are related and should be in the same partition, and thus grouped together in storage (could be coaligned with another entity, with time, with a value, with a batch)
4	partition	See https://docs.datomic.com/on-prem/schema/schema.html#partitions Partitions group data together (in storage), providing locality of reference when executing queries across a collection of entities. Entities in the same partition to sort and be stored together in E-leading indexes, i.e. EAVT and AEVT. Partitions are associated with entity ids, and are named by keywords, or referred to by index in a space. Encoded as hi bits in entity ids Partition entity ids are suitable as arguments to d/tempid, d/entid-at, and :db/force-partition
5	explicit partition	partition associated with an explicitly-created, named partition entity datomic comes with 3 explicit partitions: :db.part/db :db.part/user and :db.part/tx
6	implicit partition	a partition that can be referred to by its index in a range of integers 0<=x<524288. These partitions have entity ids, and they require no explicit installation. Their entity ids consist of: part=index with the 20th bit set, eidx=0 In larger applications, you may want to spread data across a larger number of partitions. Implicit partitions provide a mechanism for this. Implicit partitions provides a way to manage a large number of partitions numerically and algorithmically. old ref to partition sharding
7	primary	the owning side of affinity, use to choose partition for related (e.g. the customer)
8	related	the "owned" side of affinity, gets partition from primary (e.g. some activity entities related to a particular customer)

Questions

a most powerful thinking tool

to formulate a question is to reify what you seek

getting questions right is half the battle

questions provoke, often novel thinking

logic (just) helps us rule out some of it

Technique: The Socratic Method

interrogate *'ask together'*

examine an idea dispassionately
questioning its underlying assumptions, consistency

Dispassionate *'without suffering'*

you are not your idea

you are a source of ideas, some better than others

We don't define/opine the truth, we discover it

'The Socratic Method: A Practitioner's Handbook' - Farnsworth

Father Watson's Questions

Where are you at?

Where are you going?

What do you know?

What do you need to know?

Devs are good at the first two, but those miss *'why?'*

Technique: Reflective Inquiry

	Understanding	Activity
Status *'to stand'*	What do you know?	Where are you at?
Agenda *'to be done'*	What do you need to know?	Where are you going?

this is a framework that can be applied throughout the design process

note the importance of thinking about your thinking

reflect - *'bend back'*

inquiry - advancing knowledge, is the driver

Technique: PM Top Story/ticket

Several design techniques contribute to your 'top' story in PM

Looking to always create structured stories with sections for:

- Title
- Description
- Problem Statement
- Approach

Design stories contribute to **building a 'top' story**

Example: Story

Support Java Streams in Clojure's seq functions

Description

As Java Streams become more pervasive, users struggle with being unable to process them using Clojure's standard library, which does not accept them.

Problem

Java Streams are not seqs, nor do they implement any interfaces to which Clojure already bridges, thus are not accessible to Clojure's functional operations. Furthermore, they are stateful and not functional, and require special handling.

Approach

Java streams are stateful (like iterators) but we need the ability to seq (like `iterator-seq` which caches from stateful iteration), reduce, and into from a stream. Once we have that, we can leverage existing Clojure seq/transducer tech to manipulate streams.

Create:

- Reduce support via Stream.reduce, needs BinaryOperator (see functional interfaces story)
- `stream-seq!` similar to `iterator-seq` - creates a seq as it reads stream
- `into` support via new `stream-into!` - implemented with Collector, and utilizing transients etc
  Note these will be 'terminal' functions on the Stream.

Planning Sheet: https://docs.google.com/spreadsheets/d/1gmVNHCa6 3dy_-TcE/edit#gid=1073327933

Design Progress

measured by increasing understanding
of the truth of the world
and your opportunities within it

decisions made **and why**

not checking off some process/method or design artifact list

or making a plan from your first idea

Design Phases

not everything with any linearity is a 'waterfall'

nor do you want 'iterative development'

iterate == Latin for *'do-over'*

better: incremental - *'grow into'*

more like a hike up the (understanding) mountain, not always up, but trending up

being able to name phase *'appearance'* helps with *'where are you at?'*

not monotonic - ok! **stay open-minded**

this is when change is cheapest

be explicit about backtracking

Phases

“these are words with a D this time”

Describe (situation)
Diagnose (possible problems)
Delimit (the problem you are going to solve)
Direction (strategy, approach)
Design (tactics, implementation plan)
Dev (build it)

at any time:

Decide (to do, or not)

Phase: Describe

the situation
bug/failure reports
feature requests, external and internal (backlog)
context

What do you know? something seems wrong/obstructive in the world
What do you need to know? the extent of it

Where are you at? observing, listening
Where are you going?
- initial story title
- **write down** a Description in top story

Technique: Description

one paragraph summary
situation/context
symptoms/reports/observations
requests

don't:

say what the problem is

accept as facts assertions that imply what the problem is

instead: X says Y

Phase: Diagnose

'know across' **possible problem(s), of two kinds**

1 - bugs/defects
- yes bugfixes need design (or revisions of a design)
- lest you just play symptom/code whack-a-mole

2 - features

What do you know? the symptoms/context
What do you need to know? the cause(s)

Where are you at? have good description, evidence
Where are you going?
- applying logic and experimentation
- to explicate *'unfold'*

Diagnose: Bugs

symptom → possible problems → (likely) problem

hypotheses (more than one)

pick one (how?)

- use logic first (to rule out)
- 'most likely' (intuition)
- makes the problem space smallest (divide and conquer)

Use the scientific method

Technique: Scientific Method

out of scope for this talk

formulate a supporting/refuting conjecture

design an experiment

write result template first
- "if this sheet were filled in we'd know X"

code it, conduct it

apply conjecture logic, repeat

Diagnose: Feature Requests

feature: factura: *making*, **of an answer**

not the problem

'we don't have feature X' is never a valid problem statement

recognize and kill all such statements

feature → problem(s) for which that feature is (one possible) answer

what is the user's intention/objective? (not how)

what is in the way?

Phase: Delimit

the problem you are going to solve
you might discover multiple problems or bigger problems during diagnosis

What do you know? what the problem is
What do you need to know?
-how to state it succinctly
-its scope

Where are you at? have diagnosis
Where are you going?

making the problem statement

Technique: Problem Statement

Succinct statement of unmet user objectives and cause(s)
not symptoms/anecdotes/desires
not remedy/solution/feature - challenge is to filter out

modify your top story title from symptom→problem
add Problem after the Description in the top story - link to diagnosis work

subject to refinement
- as your understanding increases
- don't let your problem statements get stale

This is the most important artifact you will have
if you don't relentlessly focus on a problem you may make something that doesn't solve any problem

Phase: Direction

strategy, approach
User's intentions and objectives
High-level approaches to addressing
e.g. in-proc/out, lib/app, buy/build, modify/add, automatic/manual etc

What do you know? what the problem is
What do you need to know?
- the user objectives in more detail
- the possible approaches
- the best of these
- what matters in deciding (criteria)

Phase: Direction (cont.)

Where are you at? Have description and problem statement
Where are you going?

- Enumerating uses cases
- Making a strategy DM
  - criteria, approaches and tradeoffs
- determining scope
- entering Approach section on top story

Technique: Use Cases

user's intentions and objectives
in terms of **what** the user could accomplish
were the problem solved
not how (yet)
make a blank 'how' column for later

should not start with
"the user will push an orange oval button and music will play"

later you will fill in the 'how' column with that kind of recipe for using the solution you've designed

Template: Use Cases

	A	B	C
1	problem this sheet is about	How (given solution design)	Notes
2	user intention/objective		
3	another user intention/objective		
4	another user intention/objective		
5	...		
6			
7			
8			
9			
10			
11			
12			
13			
14			

Example: Use Cases

	A	B	C
1	Morse setup and invocation	how	notes
2	I want to use Morse to develop and inspect my process	take morse as a dev-time dep in :dev alias require and use in REPL (m/inspect X) in REPL, shows up in Morse	
3	I want to use Morse but my process is remote	take replicant server as a dep in the remote process start server connect	the way to see stuff in your REPL interactions it to bind them in user ns and eval them in Morse editor
4	I want to use Morse but my process is remote and it can't or doesn't take replicant as a dep	prereqs: remote proc has a socket REPL available, and Clojure 1.12.0-alpha3 + THEN connect editor to socket REPL (or nc) (add-lib 'org.clojure/data.alpha.replicant-server) in your REPL (require replicant server as r) (r/start-replicant port) server with port in terminal launch Morse as a tool pointing to replicant-server's port	the way to see stuff in your REPL interactions it to bind them in user ns and eval them in Morse editor
5	I want Morse to handle my REPL interactions and display their results automatically	For stream repl, could re-enable this recipe using stream I/O (with downsides of not working for nrepl etc): use project classpath (-M or -X, not -T) run ui with in-proc mode use the morse/repl as the proc	NOT YET or undoced?
6	I want to connect Morse to a remote process and I want m/inspect in REPL	NOT CURRENTLY SUPPORTED	(r/start port) -> conn or magic (r/inspect conn? val) does push?
7	I want Morse to cooperate with my REPL to display results	NOT CURRENTLY SUPPORTED	clojure.main allows callback for eval?

Technique: Decision Matrix (DM)

a (google or other live-editing) sheet

A:1 what decision are you trying to make, for which problem?

Approaches - Columns (but first labels rows)
Criteria - Rows (but first labels columns)
Aspects - Cells

sheets > docs

prose docs create a linearization that makes contrast difficult

Template: DM

	A	B	C	D	E	F
1	Problem and decision this sheet is about	current approach (if there is one)	another approach	another approach	...	Notes
2		additional succinct summary	more			
3	a criterion	aspect - how this approach handles this criterion	aspect	appealing aspect		lorem ipsum
4	another criterion	aspect	blocker aspect	aspect		
5	another criterion	not so great aspect	not so great aspect	unknown aspect?		
6	...					
7						
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						

DM Columns: Approaches

first row or two describe approach
- must give you shorthand for talking, yet make clear what about
- **succinct description** of approach, use row 2 if needed
- freeze the approach title/description rows

if you are modifying something, the first 'approach' should be the status quo

columns for what others have done in same situation

and your initial ideas

A DM is about **creating** a great approach, not merely shopping

the answer is often an approach you don't begin with

DM Rows: Criteria

'means of judging/deciding'

First column - **succinct descriptions** of criteria (freeze this column)
Include criterion iff salient or relevant, sort by importance, distinction

will usually include rows for

- fitness for solving the problem (from use cases)
- various '-ilities'
- costs (time, dev effort, $), risks
- compatibility, complexity
- etc - purpose built for problem (reflective)

DM Cells: Aspects

of approach per criterion

succinct description of how approach handles criterion (or doesn't)

avoid y/n/true/false/numeric-rank criteria, and in cells

avoid judgement in text, instead use (unsaturated!) cell background color

Neutral - clear

Some challenge or negative - yellow

Seems blocking or failing to address problem - red

Seems particularly desirable/better - green

DM Cells: Aspects (cont.)

Can start with just 'pros' and 'cons' rows/cells

but important to split up later

only then are criteria explicit

and all approaches judged similarly

Contrast - *'stand against'*

edges are primary triggers of perception

Example: DM

	A	B	C	D	E	F	
1	Can't use Java methods that take Java functional interfaces without using an adapter or reify. Right now, people are doing a lot of redundant verbose reifying	currently: reify to adapt in user code	extend Clojure AFn/iFnxx to some Java interfaces	common reifying adapters in util ns	single fn adapter satisfying all common interfaces	reify-like macro	insert ada
2		what people do now	implement additional curated interfaces upon AFn/AFunction just Function, BiFunction, and Predicate?	implement delegation functions for common SAMs	e.g. 'jfn'	a macro that discovers the SAM method name and emits a complete reify	Compiler e of checkca are known Adapter w SAM targe
3	example of use	(reify AFnInterface (a-method [x] (f x))	f	(supplier f) (unary-operator f) ...	(jfn f)	(SAM java.util.Comparator [x y] (and (vector? x) (vector? y (compare (count x) (cou	f
4	Does it handle methods taking JUFs?	yes, via reify	yes, by changing fns to be JUFs	yes, via adapter (reify)	yes, via adapter (reify)		yes, by ch adapt fns
5	Does it handle methods taking primitive JUFs?	yes, via reify - but you need more primitive type hints and it makes the incantation longer	yes, but it's a bunch more interfaces to implement	yes, via adapter (can handle primitive hints)	yes, via adapter (can handle primitive hints)		yes, presu
6	syntax concerns	repetitive	like with strings, numbers, etc - no wrappers	need to know many adapter names	only one adapter fn to know	this is only marginally winning over reify (don't need method name)	
7	runtime perf	wrapper object and delegation	possible perf impacts for all fns due to more superinterfaces of AFn (type pollution)?	wrapper object and delegation	wrapper object and delegation		
8	impl impact	none	changes to IFn/AFn	10-40 adapter fns?	1 adapter fn	macro entirely in userspace	Compiler; additional IFn

DM: Tips

Avoid
the all-green column - are you rationalizing?
undistinguished columns - find the differences that matter
exhaustive or template rowsets - s.b. specific criteria, not just characteristics
links as primary cell content - ok as supplement to summary text in cell
hidden comments/popups etc - keep things in view
phrasing criteria as questions - clash with inline questions

include questions as soon as they arise!

put '?' anywhere (approach/criterion/aspect)

- if you are unsure of importance

- or the info is unknown

DM: Outputs

a succinct **description of the problem/decision** being taken on
a set of **several approaches**, succinctly described
an explicit and clear expression of **what matters in making the decision**
detailed aspects for all of the approaches per criterion
- aligned for contrast

at-a-glance, fine-grained **subjective assessment**
- subjectivity all in one place (cell color)

a set of questions for follow up

clear benefits+tradeoffs

DM: Benefits

come back later/arrive late - (re)load context

live group thinking tool - make everything visible as text
- vs voice + independent notes

promotes shared understanding
- call out ambiguity, inconsistency etc
- raise and capture questions and ideas immediately

birthplace of abstraction

provocation for background thought
hammock, sleep
where new columns and best answers are born

Phase: Design

tactics, implementation plan
the blueprint-like design

What do you know? the problem and the direction we are taking to solve
What do you need to know?
- the possible implementation approaches
- the best of these
- what matters in deciding
- how the users will use your solution

Phase: Design (cont.)

Where are you at? Have use cases and strategy/direction DM
Where are you going?

implementation approach DM(s)

design (plan) diagrams

implementation decisions

add detail to Approach section of top story

fill in 'How' column in Use Cases

how user can accomplish using feature/API etc

possible scope adjustment or backtracking if impl poses new challenges

Example: Impl DM

	A	B	C	D	E
1	Need Java experience to access Math via static functions Direction Not generic - just j.l.Math	current Math interop	Static imports - gen when needed	Code gen a Clojure ns usable at runtime	Hand code a clojure.math wrapper
2	gen when?		loading forwarding ns	build?	
3	Usage	(Math/sqrt ...)	(require '[clojure.math :as math]) (math/sqrt ...)	(require '[clojure.math :as math]) (math/sqrt ...)	(require '[clojure.math :as math]) (math/sqrt ...)
4	Definition	none	Once dynamically, when loaded or once during build if compiled	Once during Clojure build or ahead of time and check into git? once	Once
5	Load cost	n/a	Uses reflection to gen at load For all java.lang.Math: ~3 ms Or at compile time for normal	Normal ns load time (~ 0.1 ms)	Normal ns load time (~ 0.1 ms)
6	Runtime perf	Fast	Fast via inlining	Fast via inlining	Fast via inlining
7	docstrings?	no	yes, but just point to the wrapped method (can't easily get to actual text)	yes, but just point to the wrapped method (can't easily get to actual text)	could be whatever we want
8	Can solution be applied to other classes with static fns	yes - use static method interop	yes - easy to apply to other classes dynamically	maybe?	no
9	Apply-able	no - would need compiler changes to make static fns apply-able	yes	yes	yes
10	JDK version impact	use what you have	use what you have	depends on JDK used at build time	depends on hand coding
11	"Findability"	need to know JDK + Java interop	need to know Clojure API what API? http://clojure.github.io/clojure/	need to know Clojure API	need to know Clojure API

Technique: Diagrams

details out of scope for this talk

important complement for tables and prose, better for:

- architecture
- flows
- relationships
- representations/layouts
- UI

diagram your problems, not just your solutions

Phase: Dev

build it

You **understand why** you are making the thing - solving this problem
You **know how** to make it - few or no unknowns
You are **confident it will work**
- lots of supportive material
- keeps you on track
- facilitates adding others to team

the solution will be smaller and more general due to having designed it

Have at, with your dev toolkit and techniques

but don't build something on the same day you think of it

Thanks!

Dan (for all the notes), Stu, Alex and my other Socratic ~~victims~~ friends on the Clojure and Datomic teams

Inspiration exists, but it has to find you working

— Pablo Picasso